第18回

ソフトウェア向けの解析・検証ツールとハードウェア検証

――プログラム・チェッカをシステム設計記述の検証に利用する

藤田昌宏

ここでは，「プログラム・チェッカ」と呼ばれる，ソフトウェアとしてのCプログラムを検証するツールを，C言語ベースのシステム設計記述に適用した場合の問題について解説する．このツールは，各文の間の依存関係を解析することにより，構文のエラーや状態のデッドロック，並列プロセスのアクセス競合，バッファ・オーバフローなどを検出できる．（編集部）

本連載ではこれまで，ハードウェア設計記述の形式的検証手法について解説してきました．設計効率化のため，現状のRTL（register transfer level；レジスタ転送レベル）記述より上位の設計で，C言語をベースとした開発手法の導入が進みつつあります．C/C++で記述された設計に対する検証手法としては，ソフトウェアとしてのCプログラムを検証する手法の応用が考えられます．

今回はこの観点から，ハードウェアのC言語ベース設計記述の検証について考えていきたいと思います．

● ソフトウェア・プログラムとしてのC記述を解析

ソフトウェア・プログラムの解析や検証に，形式的手法を適用する研究は，プログラムというものが世に出て以来，ずっと続けられています．プログラムの正しさが重要なソフトウェアの例として，例えばLinuxやWindows XPなどのOSがあります．これらのソフトウェアは数百万～数千万行の規模であり，これらにも適用可能な手法であることが実用上は重要になります．したがって，これまで説明してきたようなモデル・チェッキング（model checking）の技術をC言語のプログラムに適用することは，研究レベルでは活発に行われていますが，実用的に使えるツールという観点からは，あくまでも「将来の技術」という位置付けになります．

ソフトウェア開発の世界では，そのような一般的な仕様としての「プロパティ」をチェックするプロパティ・チェッカではなく，正しいプログラムとして満たすべき一般的性質をチェックする，「プログラム・チェッカ」と呼ばれるツールが広く利用されるようになってきています．

このプログラム・チェッカで調べられる性質は，C/C++でハードウェアを記述するCベース設計においても満たすべきものであり，とくにハードウェア・ソフトウェア協調設計では，プログラム・チェッカは利用価値の高いツールとなっています．そこで以下では，現在，すでに広く利用されている，あるいは製品化されているツールを中心にプログラム・チェッカの機能を紹介し，C/C++によるハードウェア設計記述への適用が可能であることを説明していきたいと思います．

● Lintツールは構文のエラーをチェック

もっとも基本的なプログラム・チェッカとして，Lint（リント）ツールがあります．Lintは，おもにC言語で書かれたプログラムに対して，簡単な構文（シンタックス）の解析を行うことで，あらかじめ登録している満たすべき性質に反している部分がないかどうかをチェックします．

例えば，型（タイプ）の一致していない関数呼び出しや使われていない変数，初期化されていない変数を使っている箇所，まったく実行されないコードなどを検出できます．

KeyWord モデル・チェッキング，ハードウェア・ソフトウェア協調設計，Lint，プロパティ・チェッキング，SpecC，デッドロック，alloc，SAT問題

また，ある所では関数の戻り値を使っていて，ほかの所で同じ関数を呼び出しているのに関数の戻り値を使っていないとか，逆に一つの関数の中で，ある所では意味のある値を`return`しているのに，ほかでは値を戻さずに`return`しているような場合を検出することもできます．さらに，定義されているのに使われていない関数も検出可能です．

これらの機能のいくつかは，コンパイラでも検出可能であり，Lintはコンパイラのチェック機能をより強化したものであると言えます．Lintは，おもに記述を簡単に解析するだけで処理できるように作られており，その意味で大きなプログラムでも実用時間内に処理することができます．

一方，簡単なチェックで処理しているため，あやしいところはすべて警告を出すことになります．結果として，多数の警告が出され，そのうちの大多数は実はほんとうのエラーではないということもよく起こります．出力される警告のうち，プログラマにとってむだな警告を少しでも減らすため，プログラマ自身が検出するべき箇所を指定できる機能も備わっています．しかし，一般的には，プログラマにとって，あまりにも多くの警告が出されるため，出力された警告が重要なものかどうかの判定をプログラマが行えないというケースもあります．あるいは，あまりにも多くの警告があるため，その中の少数の重要な警告をプログラマが見逃してしまうといったことも起こりえます．そこで，むだな警告を少しでも減らすための研究が長年続けられており，実用ツールとなって製品化されています．以下では，それらについて説明していきます．

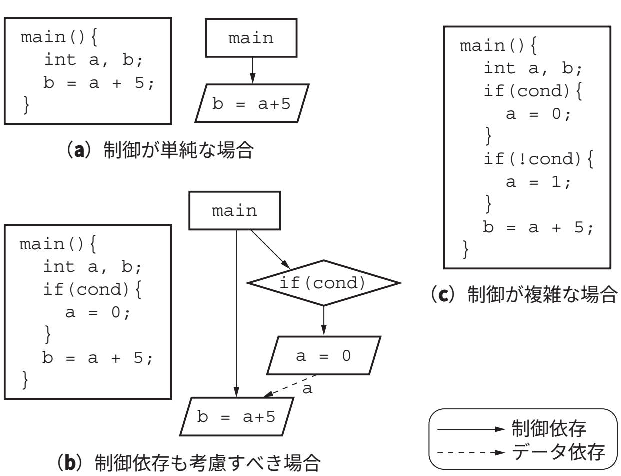


図1　変数の非初期化のチェック

変数の値を利用する前に，かならずその変数を初期化する必要がある．これは，制御依存とデータ依存の関係を部分的に調べることで判定できる．

● 各文の間の依存関係を現実的な時間で解析

一般に，プログラムの中の各文（あるいは，各文中の変数）には，データ依存，および実行順に関する制御依存があります．また，C/C++をなんらかの方法で拡張し，並列処理を記述している場合には，並列に実行される文の間の実行順を制御するためのイベント信号依存もあります．

これらの依存関係は，与えられたC/C++記述から，コンパイラ技術などを利用することで，プログラムの行数の1.5乗程度に比例する時間で解析することができます．このため，比較的大規模なプログラムでも現実的な時間で処理が完了します（処理量が爆発しない）．

プログラム・チェッカの基本は，これらの依存関係を解析することで，さまざまなプロパティが満たされているか否かを調べます．その際に，依存関係全体を解析するのではなく，可能な限り，部分的に解析することで，解析時間の短縮を図っています．プログラム・チェッカは任意のプロパティを検証するものではなく，プログラム記述にとってかならず満たすべきプロパティのいくつかを自動的に検証していくものです．検証できる典型的なプロパティとして，以下のものがあります．

- **変数の初期化が行われているか**
- **配列へのアクセスが配列の大きさの範囲内か**
- **nullとなっているポインタにアクセスしていないか**
- **並列処理の場合，共有変数へのアクセス競合はないか**

基本的な例で説明しましょう．変数の値を利用する前にかならずその変数を初期化するのはプログラミングの基本であり，すべてのプログラムが満たすべき性質であるといえます．これは，**図1**に示すように，制御依存とデータ依存の関係を部分的に調べることで判定できます．

図1(a)の場合，変数`a`が利用されている箇所の前に変数`a`への代入があるか否かを調べれば，変数の初期化をチェックできます．この場合，その変数`a`へのデータ依存はありません．つまり，初期化されていないということがわかります．また，**図1(b)**の場合，同じ変数`a`が利用されている文へのデータ依存がありますが，さらに制御依存をたどっていくことで，それは`if`文の中で行われていることがわかります．したがって，この場合，変数`a`は初期化される場合もあるし，されない場合もある，つまり，バグの可能性があると判定されます．

● 解析時間と解析の正確さにはトレードオフの関係がある

さて，ここで重要なことがあります．それは，プログラム・チェックを行う場合，どの程度依存関係を解析するかです．**図1(a)**の場合，変数aを使っている文へのデータ依存はまったくないので，変数aが初期化されずに利用されていることは明らかです．しかし，**図1(b)**の場合，変数aが利用されている文へのデータ依存は存在します．したがって，このまま解析を終了すると，変数aは初期化されていると判断してしまいます．しかし，データ依存が出ている文の制御依存をさらに調べていくと，if文の中にあることがわかるので，変数aの初期化はif文の条件しだいということになります．

さらに**図1(c)**を見てください．この場合にも，変数aの初期化はif文の中にあるので，条件しだいということになります．ただし，**図1(b)**と違って，二つのif文の条件を解析すると，すべての場合を網羅していることがわかり，実際にはいつも初期化されていることになります．

このように，正確に解析するためには，if文などを考慮する，つまり，プログラムの実行パスを考慮して解析する必要があります．また，関数の場合，その関数がどの文から呼ばれたかにも解析結果は依存します．これらを完全に考慮するとなると，厳密には，プログラムの先頭からすべての実行パスをたどりながら解析することになり，依存関係を部分的に解析するというわけにはいきません．このようにすると，通常のプロパティ・チェッキング（モデル・チェッキング）と同じ手間がかかり，大規模プログラムには適用不可能です．

そこで，実際のツールでは，可能な限り，部分的な解析で済ますということと，解析の正確さに配慮してプログラムの実行パスを考慮するということのトレードオフを検討しながら処理していくことになります．実際，検証するプロパティを制限することにより，プログラムの実行パスをかなり考慮できる手法が，商用ツールでも実現されています．

● デッドロックや並列プロセスのアクセス競合を検出

プログラム・チェッカにとって，並列動作を伴うプログラムの検証も重要な目的であり，処理が進まなくなるデッドロックの検出や，並列プロセス間の共有変数へのアクセス競合のチェックなどが，対象とするプロパティになります．この種のプロパティの検証は，一般のモデル・チェッカが扱うものですが，モデル・チェッカが初期状態からすべての可能な状態を探索する（プログラムのことばで言うと，プログラムの最初からすべての実行パスについて，すべての可能な変数の値について調べる）のに対して，プログラム・チェッカは可能な限り，部分的な解析を行います．結果的に大規模プログラムにも適用可能である場合が多いのですが，その一方で，解析の精度がプロパティ・チェッカに比べると落ちることになります．

例を**図2**に示します．通常のC/C++のままでは並列動作を記述できないので，**図2**ではSpecC言語で表現しています．SpecCでは，par文を利用して並列プロセスを起動します．また，並列プロセス間の実行の順序付けを行うために，通常，イベントを利用した通信として，wait文やnotify文が利用されます．wait(e)という文は，noity(e)という文が実行されるまで実行を待ちます．一つのプロセスでwait(e)の文が実行され，もう一つのプロセスでnotify(e)の文が実行されるとすると，wait(e)以降の文の実行は，かならずnotify(e)以前の文の後に実行されることを保証することが可能です．これは，共有変数へのアクセス競合の回避に利用されます．

しかし，notify文とwait文が適切に配置されていない

```
event e;
int y, z;
bool cond;

behavior B1(){
  void main(void){
    wait(e);
    z = 2 * y;
  }
};

behavior B2(){
  void main(void){
    y = 5;
    if(cond)
notify(e);
  }
};

behavior Main(){
  B1 b1;
  B2 b2;

  void main(void){
    cond = 0;
    par{
      b1.main();
      b2.main();
    }
  }
};
```

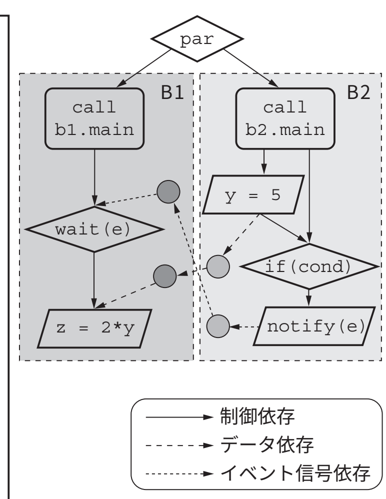


図2　並列処理についてのチェック

SpecC言語では，par文を利用して並列プロセスを起動する．また，並列プロセス間の実行の順序付けを行うために，イベントを利用した通信として，wait文やnotify文を利用する．

場合，例えばwait文はかならず実行されるが，notify文が実行されない可能性がある場合には，デッドロックが起こるケースがあることになります．**図2**の場合，notify文とwait文の間の依存関係を計算し，**図1**の場合と同じように制御依存について解析することで，そのような場合が起こりうるかを解析することになります．

一般に，並列に動作するプログラムの間の共有変数へのアクセス競合や処理のデッドロックの可能性判定を行うことは「同期検証」と呼ばれています．ちょっとしたミスでこの種のバグが入り込むのが並列プログラムですから，実用上は非常に重要な技術であるといえます．プログラム・チェッカでは，制御依存，データ依存，それにwait文やnotify文に関係するイベント信号依存を部分的に解析することで，かなりの場合について，大規模プログラムのチェックを行うことができます．

● 値の範囲の制限からバッファ・オーバフローを検出

セキュリティ上の重要なプロパティとして，バッファ・オーバフローの検出があります．これは大きなバグで，ソフトウェアだけでなく，ハードウェアを実装する場合も，オーバフローを起こすと処理を続行できなくなります．バッファ・オーバフロー（オーバラン）のチェックについて，ここでは，C言語でよく起こる文字列の扱いについて考えます．

まず，解析に利用する変数を定義します．alloc(string)という表現で，変数bufに割り当てられたバイト数の下限と上限を表現します．また，len(buf)で変数bufが実際にもつ文字列のバイト数の下限と上限を表現します．実際に利用できる領域は確保してある領域内でなければなりませんから，以下の関係が成立しなければなりません．

alloc(buf) ≦ len(buf)

より正確に言うと，alloc(buf)の値の範囲が[a,b]であり（alloc(buf)の下限がaで，上限がb），len(buf)については[c,d]である（len(buf)の下限がcで上限がd）とすると，b≦cならオーバフローすることは決してありません．また，a＞dであるとすると，いつもオーバフローすることになります．値の範囲である[a,b]と[c,d]がオーバラップしていると，バッファ・オーバフローする可能性があります．

このように値の範囲に対する制限をプログラムから抽出して，値の範囲にオーバラップがあるか否かをチェックすれば，バッファ・オーバフローが起きる可能性について調べることができます．

例として，**リスト1**のプログラムを考えます（説明のために，各行の最初に番号が入れてある）．このプログラムの各行から抽出される制限は，**リスト2**のようになります（最初の番号がプログラムの対応する行番号から得られる制限であることを示している）．5行目では，エラー・メッセージの分だけ，dstがbufより大きくなければなりません．

まず，1行目と4行目ですが，これは変数の宣言から，その変数に割り当てられた領域に関する制限が表現されています．それぞれ128に固定されます．また，2行目では，fgetsにより標準入力から文字列を読み込んでいるので，変数bufの実際の文字列の長さは，最小1，最大128となります．5行目では，bufにエラー・メッセージを追加してdstに出力しているので，上のような関係が成立します．**リスト2**の四つの制限がすべて満たされるという条件で，各文字列変数についての，実際の文字列の長さと割り当てられた領域が満たすべき関係は，以下のようになります．

len(buf) ≦ alloc(buf),
len(dst) ≦alloc(dst)

これがいつも成立するかどうかをチェックすることが，プログラム・チェッカのしごととなります．bufについては問題ありませんが，dstについては，alloc(dst)の値128をlen(dst)が超える可能性があることがわかり，バ

リスト1　バッファ・オーバフローの検出

```
1:    char buf[128];
2:    while (fgets(buf, 128, stdin)) {
3:      if (!strchr(buf, '¥n')) {
4:        char dst[128];
5:        sprintf(dst, "error: %s¥n", buf);
6:        ...
7:      }
8:      ...
9:    }
```

リスト2　リスト1から抽出される制限

```
1:    [128, 128] ⊆ alloc(buf)
2:    [1, 128]   ⊆ len(buf)
4:    [128, 128] ⊆ alloc(dst)
5:   len(buf)+8  ⊆ len(dst)
```

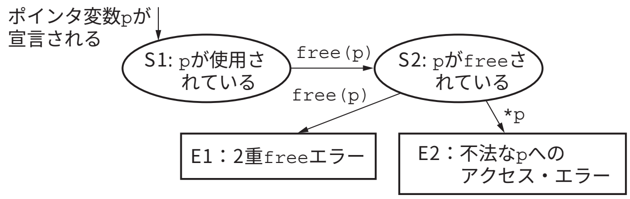


図3　ポインタ変数の2重freeのチェック
割り当てた変数の領域を開放する操作freeを2重に行ってしまわないかどうかをチェックするために，ルールを記述した．

リスト3　ルールを記述してプロパティをチェック

```
1:    int *p;                        // S1
2:    p = malloc(sizeof(int));       //
3:    *p = 7;                        //
4:    free(p);                       // S2
5:    if (...) {                     //
6:      *p = 110;                    // E2
7:    } else {
8:      free(p);                     // E1
9:    }
```

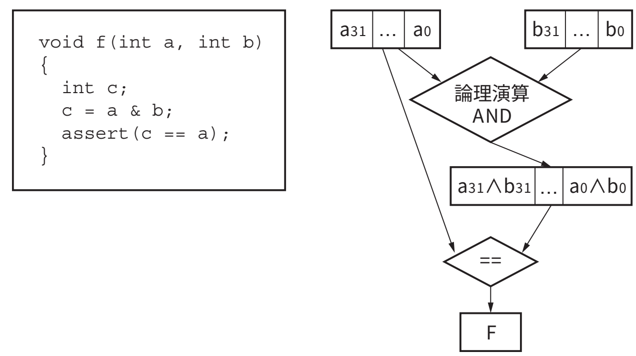


図4　CプログラムのSAT式への変換
各整数変数に対して，そのビット幅である32個の論理変数を割り当てれば，プログラムの記述を右のように論理式に変換できる．assert文が成立するか否かの判定は，Fが1'になるかならないかで判定できる．

ッファ・オーバフローの可能性がプログラム・チェッカによって指摘されます．

以上の例は制御が簡単な場合でしたが，制御が複雑な場合には，ある程度，制御依存やデータ依存をたどって解析していく必要があります．

● ルールを記述して多種多様なプロパティを一括チェック

以上の話は，チェックするプロパティごとにアルゴリズムを考えるという立場でしたが，それではチェックできるプロパティの種類を増やすことは容易ではありません．そこで，調べたいプロパティをチェックする方法をルールの形で記述して検証できるプログラム・チェッカもあります．

簡単な例として，割り当てた変数の領域を開放する操作freeを2重に行ってしまわないかのチェックを考えます．この場合のルールは**図3**のようになります．プログラムの各実行文ごとに，このルールに対応する状態遷移図上の状態がどのように変化するかを調べていき，エラー状態に陥ればプログラムに問題があるとします．

例として，**リスト3**のプログラムを考えます(// 以降は**図3**のルールの状態を示している)．**リスト3**の6行目と8行目でエラーが起きる可能性があることが，**図3**の状態からわかります．この場合はわかりやすいまちがいなので，簡単に検出できます．制御がより複雑になると，今までと同じようにある程度複雑な解析が必要になるケースもあります．

以上のような手法を実際にツールとして製品化した企業として，米国Coverity社や米国Fority Software社，米国Ounce Labs社(Prexis)，米国Secure Software社(Code Assure)，米国Application Defense社，米国SPI Dynamics社などがあります．それぞれのツールには特徴があり，また，使いこなしのテクニックも重要なので，導入してすぐに成果を上げるにはそれなりの試行錯誤が必要です．しかし，これらのツールを利用してLinuxカーネルのバグなど，実際に利用されているプログラムのバグを多数発見できているのも事実です．とくに，**図3**のようなルールで表現されたプロパティを検証できるツールの場合には，ハードウェア設計に利用するC/C++記述に対して適切なルールを用意することで，うまくチェックできることが多々あります．

● プログラムをSATに変換・解析する手法がLSI設計へ

最後に，プロパティ・チェッカなどで利用されている論理式の充足可能性判定問題(SAT問題)に帰着してプログラム・チェックを行う手法の研究状況について，簡単に説明したいと思います．以前にも述べましたが，SATソルバは近年大きく進歩しています．これをうまく利用することで，実用規模のプログラムの検証を行おうという研究が活発に進められています．

話を簡単にするため，ここではプログラム中の変数は整数型であるとします．ハードウェア設計記述の場合，整数変数と論理型の変数だけが利用されることも多いと思います．すると，**図4**に示す例のように，プログラムは単純に論理式の関係に変換することができます．各整数変数に対

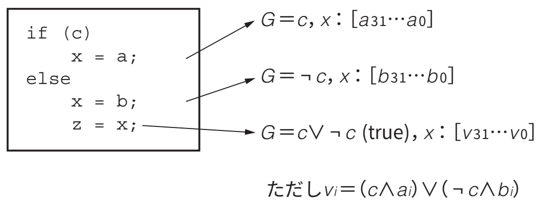


図5　制御フローの表現

プログラムの中にif文などの条件文がある場合は，ガードと呼ばれる条件を各変数の値に付け加える．ガードが'1'になる場合に対応する値をとると考える．

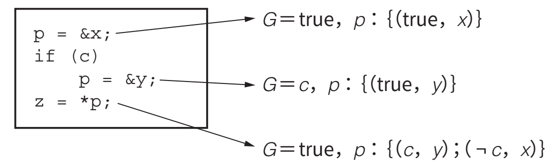


図6　ポインタの表現

ポインタが含むプログラムの場合には，各ポインタ変数がどこを指す可能性があるかを列挙する．ポインタ変数がどこを指すかについて，その場所とそこを指す条件をペアの形で表現する．

して，そのビット幅である32個の論理変数を割り当てれば，プログラムの記述を**図4**の右のように論理式に変換できます．**図4**では，`assert`文が成立するか否かの判定は，`F`が'1'になるかならないかで判定できます．

プログラムの中に`if`文などの条件文がある場合は，**図5**に示すようにガードと呼ばれる条件を各変数の値に付け加えます．ガードが'1'になる場合に対応する値をとると考えるわけです．`then`と`else`のそれぞれの場合についてガードが付けられます．そして`if`文全体としては，ガードは条件なしとなり，その代わりに`v`が条件`c`を含めて定義されていることに注意してください．

また，ポインタが含むプログラムの場合には，**図6**に示すように，各ポインタ変数がどこを指す可能性があるかを列挙するようにします．ポインタ変数がどこを指すかについて，その場所とそこを指す条件をペアの形で表現します．

以上のようにすれば，与えられたプログラムをSAT式に変換できます．そして生成されたSAT式を解析すれば，一般のモデル・チェッキングを行うことができます．ただし，この場合は，数百～数千行程度のプログラム・サイズが解析できる限界になります．そこで，**図3**に示した状態遷移図によるプロパティ解析のルールと併用することで，SAT式を大幅に簡略化して検証する手法が研究されています．その初期評価結果によると，600万行程度のLinuxソースに対して，数十のワークステーション上で実行することで，1，2時間程度でいくつかのプロパティが検証できることが実証されています．また，実際にいくつかのバグも検出できています．

このような手法は，今後，システム・レベルのLSI設計にも順次適用されていくと考えられます．

ふじた・まさひろ
東京大学 大規模集積システム設計教育研究センター（VDEC）教授

<筆者プロフィール>
藤田昌宏．1985年，東京大学大学院 情報工学専門課程修了．工学博士．同年，富士通研究所入社．論理設計用CAD技術の研究開発に従事．1993年より米国富士通研究所へ出向，CAD研究グループを立ち上げる．2000年3月より東京大学大学院 工学系研究科電子工学専攻 教授．2004年4月より東京大学 大規模集積システム設計教育研究センター（VDEC）教授．ディジタル・システム設計支援技術，とくにシステム・レベル合成，論理合成，論理検証を中心に研究している．